Case Study: Building a Serverless Messenger Chatbot

Lehvä, Jyri

Springer, Cham
2018-02-22

Lehvä , J , Mäkitalo , N & Mikkonen , T 2018 , Case Study: Building a Serverless Messenger Chatbot . in I Garrigós & M Wimmer (eds) , Current Trends in Web Engineering : ICWE 2017 International Workshops, Liquid Multi-Device Software and EnWoT, practi-O-web, NLPIT, SoWeMine, Rome, Italy, June 5-8, 2017, Revised Selected Papers . Lecture Notes in Computer Science , vol. 10544 , Springer, Cham , pp. 75-86 , International Conference on Web Engineering , Rome , Italy , 05/06/2017 . https://doi.org/10.1007/978-3-319-74433-9_6

http://hdl.handle.net/10138/237066
https://doi.org/10.1007/978-3-319-74433-9_6

unspecified
acceptedVersion

Case Study: Building a Serverless Messenger Chatbot

Jyri Lehvä, Niko Mäkitalo, and Tommi Mikkonen

University of Helsinki,
Department of Computer Science,
Helsinki, FINLAND,
{jyri.lehva,niko.makitalo,tommi.mikkonen}@helsinki.fi

Abstract. Major chat platforms, such as Facebook Messenger, have recently added support for chatbots, thus making chatbots more accessible for the end users. This paper presents a case study on building and designing a Messenger chatbot for a media company. The chatbot uses a Serverless Microservice architecture which was implemented using Amazon Web Services (AWS) including API Gateway, Lambda, DynamoDB, SNS and CloudWatch. The paper presents the architecture and reports the findings regarding the design and the final implementation. These findings are also compared the to other recent studies around the same emerging topic.

Keywords: Chatbot; Internet Bot; serverless computing; microservices; AWS; AWS Lambda; Facebook Messenger;

1 Introduction

Companies such as Facebook, Google, Apple and Amazon have recently started promoting conversational UIs such as chatbots and bots that can be commanded with voice and text [18]. That has resulted in growing interest interest regarding how such technology could be used to improve business in many domains.

In this paper we report a two-month case study where a chatbot was built for a media company in early 2017. The goal of the project was to design a scalable, modern architecture for a chatbot that follows liquid software principles [14]. Hence in the paper we present such an architecture and explain the design behind our solution. We also report the findings and experiences gained from building the given solution. The chatbot has been beta tested with 150 real users but is still not launched to production by the time of writing this paper.

The chatbot was built for Facebook Messenger platform [7], which lets developers to build chatbots that are directly available in the widely used Facebook Messenger applications. The architecture followed the Serverless Microservices approach and was built using Amazon Web Services [9,15,11,12]. Serverless platforms are said to have many benefits compared to server-based approaches; The infrastructure used for running the serverless platforms are managed by the cloud provider which removes the need to worry about server management, which is

an inexpensive solution, and which can scale up rapidly [10,13,16]. While these are great benefits, some studies say that serverless services can be hard to debug [13,17]. Thus, in addition to designing the architecture and building a chatbot, this case study also tries to find out if these benefits apply to the project, and if the implemented serverless service is hard to debug.

The rest of the paper is structured as follows. In Section 2 we describe the case study. In Sections 3 and 4 we introduce the used technologies. In Section 5 we describe the designed architecture. In Section 6 we report the results of the case study, and in Section 7 we discuss about the results. Finally, in Section 8, we outline some future work and draw some final conclusions.

2 Case Study: Chatbot for a Media Company

The media company's vision of the chatbot was to build an assistant that could help the users to follow up the latest news of their interests easier. Together with the company, we analyzed and defined the chatbot to have a set of features.

Major features of the chatbot included the following:

- Customizable dialogues
- Read latest news
- User interface for reading latest news
- Order customizable news packages
- Select delivery time for the news packages
- Receive breaking news as they happen
- Subscribe for a weekend news package with a hardcoded delivery time

Other requirements included a possibility for adding natural language processing (NLP) features later on, as these were left out from the project's first phase. The idea of these NLP features is to try finding users' intents from free text input and then trying to react with a proper reply. The architecture also had to have a support to add other chat platforms such as Slack or Telegram later if so wished.

Technical requirements were found out when more carefully analyzing the major features that were requested by the media company. The main requirements are:

- The chatbot had to have a data structure or format for the dialogue. For that the team decided to use JSON notation where the dialogue is written as so called states. Each state had an id, message sent to the user and info about the next state to follow.
- There had to be a way to persist the state of the user between messages. The chatbot could not simply be 100% stateless. The reason for that was that there's a bunch of so called chat flows where the first step leads to a

second step and so on. The chatbot just had to know from which step the user is coming and sometimes even what the user had selected in the flow a couple of steps backwards.
- The future state paths could also contain conditional logic that depends on the past selections. For the conditional logic, the JSON had a list of function names per state which should be called when the user enters the state.

3 Facebook Messenger Platform

Facebook Messenger is a service for instant messaging with Facebook friends. It is available as an application for all major mobile platforms such as iOS, Android and Windows. In 2016 Facebook Messenger platform was extended so that it enables building chatbots that can have conversations with people on Facebook [7].

The conversations with chatbots can be started by searching for the chatbot by the name of the Facebook page they are linked to on the Messenger. The chatbots can also be discovered through direct links that can be advertised on Facebook or with Facebooks QR-code-like system called Messenger Codes which can be scanned using the Messenger application [7].

Facebook doesn't offer any solution to build the chatbot backend. Instead, the backend needs to be built and hosted somewhere else and Facebook leaves that up to the developer to decide where and how. The platform offers callbacks which call a webhook (configured by the developer) when a user sends a message to the chatbot. That way the message gets delivered to the backend. The chatbot can answer to the message by sending it to Facebook Send API which then delivers it to the user as a reply to the initial message.

The UI of the Messenger, which is shown in the Figure 1, is completely in hands of Facebook. There is a variety of different kind of templates for the messages which can be used by the chatbots. These templates include swipeable carousels, lists, receipts, images, text, buttons, quick replies plus some other options.

4 Amazon Web Services (AWS)

AWS [5] is a cloud service platform built by Amazon company, offering a wide range of services for developing software. The services are hosted by Amazon in their cloud environment and the main idea is that the developer doesn't have to worry about traits such as availability, scalability or reliability as Amazon promises to handle them automatically depending on the needs of the developer and the application(s). The cloud service platform consists of tens of services and solutions for topics such as AI, messaging, storage, computing, game development, databases and analytics. Next, we shortly introduce some of the services that were used to build the chatbot. The exact tools we used for designing the serverless system are the following.

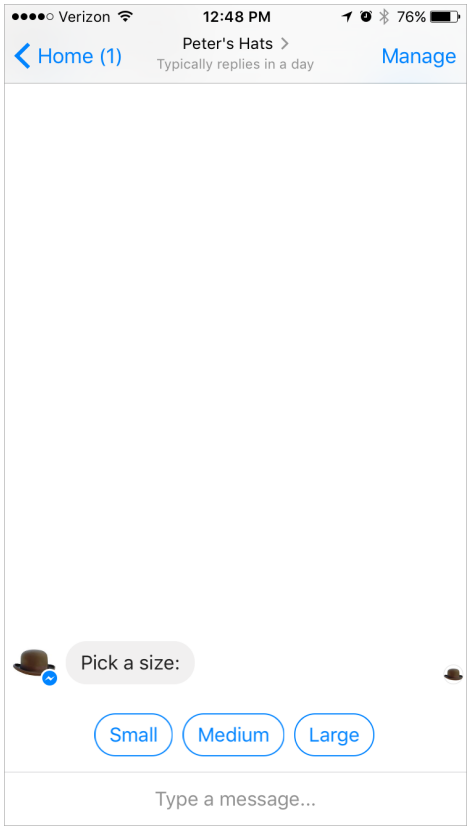

Fig. 1. Facebook Messenger UI with three quick reply buttons [7].

Lambda is the Serverless environment offered by AWS [6]. The code uploaded to Lambda can be triggered with events from Mobile apps, HTTP endpoints and other AWS services such as API Gateway, Simple Notification Service (SNS) or CloudWatch.

Simple Notification Service (SNS) is a web service that can be used to send notifications from the cloud [4]. The notifications can be send as push notifications to different kinds of clients such as mobile devices, as SMS messages, emails or HTTP/HTTPS requests. The SNS follows the publish-subscribe messaging paradigm where messages are being sent to the subscribers every time there is a new message which removes the need of subscribers from constantly polling for new messages. Benefits of SNS include option to configure retry policies if the message delivery fails for some reason, easy to set up from the web-based AWS Management Console and pricing depends directly on the number of messages being delivered through the SNS. First 1 million messages each month are free [4].

CloudWatch is a monitoring system for AWS cloud resources [2]. It provides easy way to read logs, set up alarms, monitor system performance and resource utilization and set up web dashboards to make visualizations of all those together. CloudWatch also supports making scheduled events to invoke other AWS resources such as the SNS or Lambda.

API Gateway can be used as a gateway for incoming requests for other AWS resources, such as Lambda [1]. When the requests come through API Gateway and continues to Lambda, they always take advantage of the Amazons worldwide edge locations to provide the requests the lowest latency as possible. API Gateway lets the developer also to set up caching logic to prevent some requests from hitting the backend systems at all. Other benefits include easy setup of RESTful endpoints for existing services, security controls and throttling of incoming requests during traffic spikes.

DynamoDB is a NoSQL database that supports both document and key-value store models [3]. It promises a bunch of features such as seamless scaling, high availability, secondary indexes, free-text search, strong consistency and cross-region replication. It's easy to set up and Amazon provides web interface to browse the tables in the database.

5 Towards Serverless Chatbot Architecture

The main motivation for serverless computing approach is that the developer doesn't need to manage servers. Instead, the developer can simply just upload the code (e.g. a function) to the serverless environment where the code gets executed when it's triggered by some event. Thus the developer is free from worrying about security updates or any other work related to keeping the environment up and running as all that work is outsourced to the service provider [9,5,10,13].

The serverless environment will do scaling automatically depending on the workload, and at times, when there's zero workload, it won't use any resources at all [10]. That can make serverless computing very cost efficient as the customer only pays when the code is being executed. Furthermore, this can be a good motivation to optimize the code in general. As an example, reducing the duration of the code execution from 1 second down to 200ms would directly translate to 80% savings without making any infrastructural changes [10,16]. Due to not requiring system administration work, the serverless approach can also lead to cost savings in operational management [13].

Architecture of the Chatbot has been depicted in Figure 2. The user sends a message using the Messenger application. The message goes to Facebook and triggers a callback to a webhook that contains URL pointing to the API Gateway. The API Gateway passes the event to "facebookWebhook Lambda function" that saves the user id to DynamoDB and formats the message to the internal format

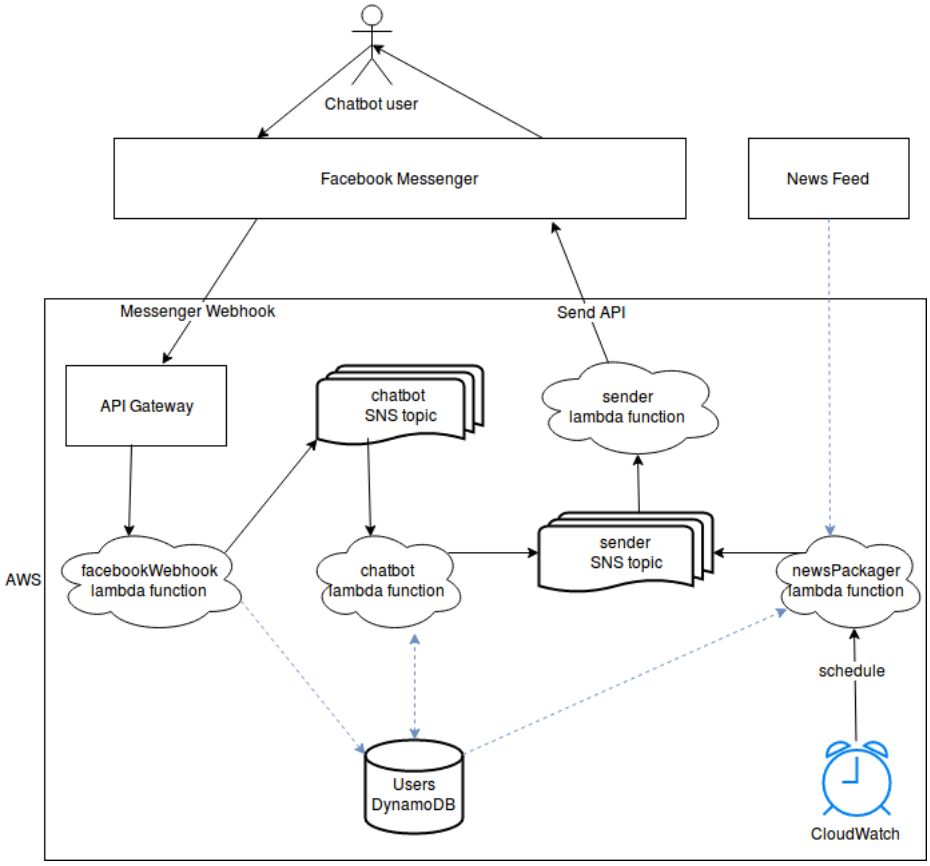

Fig. 2. Architecture of the chatbot.

used in the chatbot and passes it to "chatbot SNS topic" that forwards the event to "chatbot Lambda function".

The chatbot Lambda function contains all the business logic of the chatbot. The whole dialogue is located there as a JSON-file and the chatbot works almost as a state machine following the states found from the dialogue. The user's state is kept in DynamoDB, and if the user makes selections (e.g. selects which themes to add to the news package) those are put to DynamoDB as well.

After the "chatbot Lambda function" finishes it passes the result to "sender SNS topic" that forwards the event to the third Lambda function called "sender". The senders job is to format the reply to a proper format that is accepted by Facebook and then send it to the Facebook Send API that handles the delivery of the message to the end users Messenger Application.

The architecture also contains a component called "newsPackager Lambda function". The purpose of the function is to read news from a third-party API. This process gets invoked once every minute by the CloudWatch scheduler. If

there's users who have subscribed to news packages, they get the news delivered to their Messenger apps, or if there's new breaking news, then those gets delivered to users who have subscribed. Same logic also applies to sending the weekend news package for those who have subscribed.

6 Results and Experiences

Overall the designed architecture worked as expected. It also offers easy way to extend the system by adding other chat platforms without messing up the existing service since each task is separated as their own Lambda function. The service can be extended with other platforms simply by adding a new API Gateway, writing a new webhook and a sender using a new Lambda function for each new platform. The new API Gateway acts as a webhook for the incoming messages from the new platform and forwards them to a proper Lambda function that knows how to translate them to the format accepted by the chatbot Lambda function. Finally, the new sender would translate the replies from chatbot Lambda function to a format accepted by the new platform and send them there back to the users.

Even though the architecture consists of many different components calling each other, the latencies stayed low. One of our fears was that if each component added n milliseconds to the request time, then it could add up and make the chatbot respond too slow to the users. Luckily, that wasn't the case.

The scaling of the architecture was tested with 150 beta testers and a load test. During the beta testing there were no signs of errors about loads going too high or the Lambda invocations taking too long. The beta testing consisted of normal day to day usage and two scheduled notifications which were sent to all users at the same time trying to wake them up to use the chatbot to read news. This resulted to spikes in traffic. Lambda platform did what was promised and scaled up as needed without the developers needing to do any extra effort to achieve it.

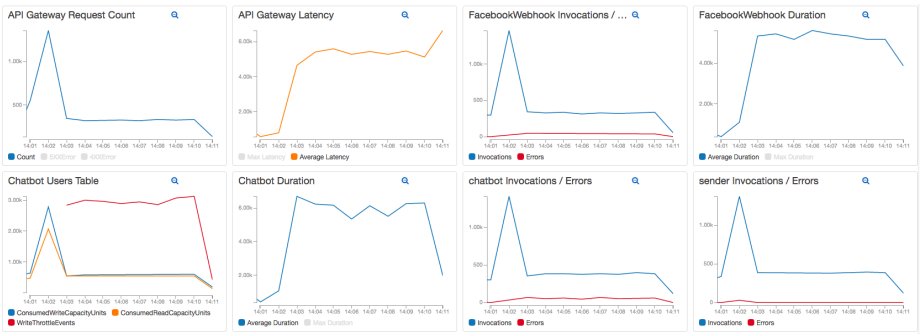

Fig. 3. CloudWatch Dashboard with throttled DynamoDB writes.

Bottlenecks of the architecture were revealed by using load testing. This was performed by using Apache Bench to call the chatbot with different amounts of concurrent connections, totaling thousands of requests during the whole testing session. To monitor what happens within the application a CloudWatch Dashboard with multiple widgets to visualize latencies, durations, throttling and invocations of the components in the architecture was set up.

It quickly became obvious that when moving to higher amounts of load compared to the loads caused by the tests with 150 real users, the DynamoDB became a bottleneck as it started to throttle the write requests, as can be seen in Figure 3. The throttling happened due to hitting the limits of the provisioned capacity units. That caused a chain reaction to the Lambda functions which started to hit the timeout limits and failed as they were pending the throttled DynamoDB calls. Also the API Gateway latencies increased considerably when the throttling happened.

After increasing the amount of capacity units for the DynamoDB and running load tests, with even higher amount of concurrent connections and requests, there were no signs of errors and the whole application was handling the requests well. This further proved that the DynamoDB and the capacity units were the bottleneck. This scenario is shown in Figure 4.

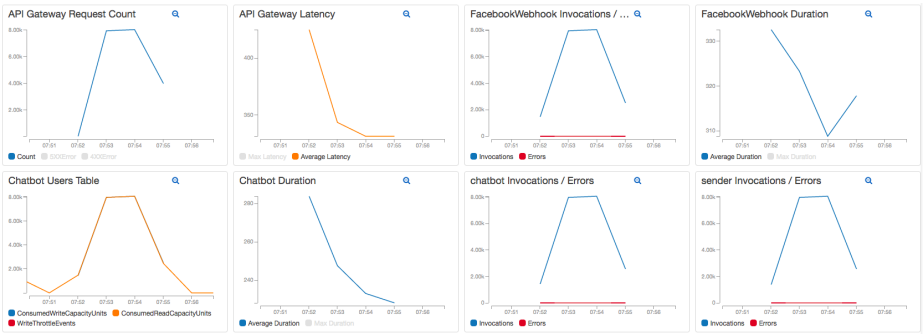

Fig. 4. CloudWatch Dashboard with increased DynamoDB capacity units.

Optimizations were made to the amount of DynamoDB calls after the tests since there was a couple of unnecessary calls being made. Many of them were replaced by adding more data to the payload sent between the Lambda functions. This removed the need of first writing something to DynamoDB and then querying it back in the second function. The tests also raised the team's attention to think how to configure the capacity units. The strategy was to increase the amount of the provisioned units for the production launch of the chatbot and then scale the amount to match the usage after monitoring the consumed capacity units over time with real users and real production usage scenarios manually after some time has passed.

The total costs of using the platform has been low. When the development time spent was about two months, there had been 2 different environments being used on the same AWS account, and the past 2 weeks had been a beta testing period with 150 beta testers, the whole cost for all that was less than one euro. A little surprise was that each invocation of a Lambda function costs 100 milliseconds – even if the invocation runs the code less than 100 milliseconds meaning that 5ms invocation counts as 100ms, and 101ms invocation counts as 200ms invocation. That might leave some room for cost optimization in the future if there's many Lambda invocations that are fast and just invoke other Lambda functions. Luckily the time spent running each Lambda function and the memory it used gets logged automatically every time the function is invoked, which makes it easy to find out if there is room for improvements of Lambda functions.

The developers' experiences about the chosen approach were fairly positive. The chatbot seemed to be a good match for the serverless computing platform as the code can be mostly stateless, and the calls to database were simple, resulting to rather fast queries. The developer team had very positive feelings about the serverless environment in general after the project and will not hesitate to use it again if there's a project where it's a good fit.

There were some problems debugging Lambda functions at first, but after paying attention to logging and learning how to run Lambda function locally using the Serverless Framework [8] debugging and development got a lot easier. The first approach to read logs was to deploy the Lambda function to AWS and then trigger it from the Messenger chat and wait for the logs to appear on the CloudWatch web UI. That turned out to slow down the development as the developer would spend a lot of time deploying and triggering the Lambda functions and then wait for the logs to update. The more efficient way was to invoke the Lambda functions with the Serverless framework on developer's local machine. That way the function can be executed on terminal in an instant and the logs are shown right away so there was no need to wait for the deployment and the CloudWatch to update the logs which lead to way more faster development cycles.

Setting up new environments was easy and fast after setting up AWS account and creating a configuration file for Serverless Framework. The configuration file describes to the Serverless Framework which AWS-components are needed and how they are linked together and then creates the environment with a single command on terminal. After that the new environment is ready to use. That was a very efficient way of setting up own environments for each developer in the team and it also worked for setting up the final production environment.

7 Discussion

Other researches and articles [10,13,16,17] pointed out that serverless microservices can be very inexpensive, require close to zero effort to manage them as

there's no need to manage servers and they are highly scalable. The findings from this case study were similar. The case study also had signs of the debugging problems but they were mostly solved by using the existing tools more efficiently. The reason why that has been an issue for the other research [17] could be explained by different platform as the research didn't use AWS. Other explanation could simply be that the tools have improved during the time after the research was released as the serverless platforms are all improving very rapidly. Though it's clear that the debugging and logging options are mostly limited to what the service providers are offering.

The beta test with 150 users gave insight on how the chatbot is used by real users. This information could be used to roughly estimate the loads and resource usage of the chatbot with different amounts of users. That combined with the results of the load tests showing the bottlenecks of the current setup was enough to point out where the focus should be when it comes down to scalability of the chatbot. While the current architecture is scalable, the limiting factor seems to be the DynamoDB and the provisioned capacity units.

As the results showed, there's no need to manage servers or make extra effort for the scaling, it could be that in the future that kind of tasks will be more and more a problem just for the platform providers than they are for the developers and architects, and that there could be less demand for that kind of expertise. These cloud platforms look more and more like a pile of different kind of puzzle pieces that require expertise to put together efficiently to form services for businesses. This lets the project teams to focus more on the business problems and the implementation of the code since there is less to worry about the hosting of the services. The flip-side of the coin is that the developers need to be aware of the pricing of the different services since this has effects on the architecture or business logic of the software.

8 Conclusions and Future Work

This case study presented an architecture for Messenger chatbot built on top of Amazon Web Services. The experiences gained from designing and building the chatbot were compared to other researches about same kind of topics. The designed chatbot architecture turned out to be extensible and scalable, requiring close to zero management and even to be cost efficient.

The future work for the chatbot application includes addition of Natural Language Processing and AI to add better support for free text input from the users. For those features, a bunch of web services will be benchmarked to find out if they can handle the needs of the chatbot. One of the biggest fear with NLP is the support for finnish language which might be quite lacking compared to more common languages like English, for instance.

Future work could also include cost optimization as the pricing model charges the Lambda invocations per 100ms of code execution time. It might sometimes make sense to run two fast scripts within one invocation instead of two invoca-

tions. Though, that should not become a problem unless the number of users increases dramatically.

Finally, trying out alternative databases and comparing them to DynamoDB from the point of view of automatic scaling should also be interesting. It might also be possible to automatically provision more capacity units for the DynamoDB by triggering events from CloudWatch as the amount of consumed capacity units changes and then programmatically increase or decrease the amount. Though there could be some latencies preventing this kind of solution from being able to address sudden spikes in the traffic.

Acknowledgements

The research was supported by the Academy of Finland (project 295913).

References

1. Amazon api gateway. `https://aws.amazon.com/api-gateway/`. Online: Checked April 9 2017.
2. Amazon cloudwatch. `https://aws.amazon.com/cloudwatch/`. Online: Checked April 9 2017.
3. Amazon dynamodb. `https://aws.amazon.com/dynamodb/`. Online: Checked April 9 2017.
4. Amazon simple notification service (sns). `https://aws.amazon.com/sns/`. Online: Checked April 10 2017.
5. Amazon web services. `https://aws.amazon.com/`. Online: Checked April 5 2017.
6. Aws lambda. `https://aws.amazon.com/lambda/`. Online: Checked April 10 2017.
7. Facebook messenger platform. `https://messengerplatform.fb.com/`. Online: Checked April 5 2017.
8. The serverless application framework. `https://serverless.com/`. Online: Checked April 5 2017.
9. A. Eivy. Be wary of the economics of "serverless" cloud computing. *IEEE Cloud Computing*, 4(2):6–12, March 2017.
10. Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openlambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'16, pages 33–39, Berkeley, CA, USA, 2016. USENIX Association.
11. C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis. Microservices in practice, part 1: Reality check and service design. *IEEE Software*, 34(1):91–98, Jan 2017.
12. C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis. Microservices in practice, part 2: Service integration and sustainability. *IEEE Software*, 34(2):97–104, Mar 2017.
13. Mike Roberts. Serverless architectures. `https://martinfowler.com/articles/serverless.html`. Online: Checked April 9 2017.
14. A. Taivalsaari, T. Mikkonen, and K. Systä. Liquid software manifesto: The era of multiple device ownership and its implications for software architecture. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 338–343, July 2014.

15. J. Thönes. Microservices. *IEEE Software*, 32(1):116–116, Jan 2015.
16. Mario Villamizar, Oscar Garces, Lina Ochoa, Harold E. Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, and Mery Lang. Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016.
17. Mengting Yan, Paul Castro, Perry Cheng, and Vatche Ishakian. Building a chatbot with serverless computing. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, page 5. ACM, 2016.
18. Mariya Yao. Does conversation hurt or help the chatbot ux? *Smashing Magazine*, November 2016.